Journal of Artificial Intelligence Research 67 (2020) 549-579 Submitted 09/2019; published 03/2020

Robust Multi-Agent Path Finding and Executing

Dor Atzmon DORAT@POST.BGU.AC.IL
Roni Stern STERNRON@POST.BGU.AC.IL
Ariel Felner FELNER@BGU.AC.IL
Dept. of Software & Information Systems Engineering
Ben-Gurion University of the Negev
P.O.B. 653 Beer-Sheva 8410501, Israel

Glenn Wagner GLENN.S.WAGNER@GMAIL.COM
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213, United States

Roman Barták BARTAK@KTIML.MFF.CUNI.CZ
Dept. of Theoretical Computer Science & Mathematical Logic
Charles University, Faculty of Mathematics and Physics
Malostranské nám. 25, 118 00 Prague, Czech Republic

Neng-Fa Zhou ZHOU@SCI.BROOKLYN.CUNY.EDU
Dept. of Computer & Information Science
CUNY Brooklyn College
2900 Bedford Ave, Brooklyn, NY 11210, United States

Abstract

Multi-agent path-finding (MAPF) is the problem of finding a plan for moving a set of agents from their initial locations to their goals without collisions. Following this plan, however, may not be possible due to unexpected events that delay some of the agents. In this work, we propose a holistic solution for MAPF that is *robust* to such unexpected delays. First, we introduce the notion of a k*-robust* MAPF plan, which is a plan that can be executed even if a limited number (k) of delays occur. We propose sufficient and required conditions for finding a k-robust plan, and show how to convert several MAPF solvers to find such plans. Then, we propose several *robust execution policies*. An execution policy is a policy for agents executing a MAPF plan. An execution policy is *robust* if following it guarantees that the agents reach their goals even if they encounter unexpected delays. Several classes of such robust execution policies are proposed and evaluated experimentally. Finally, we present robust execution policies for cases where communication between the agents may also be delayed. We performed an extensive experimental evaluation in which we compared different algorithms for finding robust MAPF plans, compared different robust execution policies, and studied the interplay between having a robust plan and the performance when using a robust execution policy.

1. Introduction and Overview

The *Multi-Agent Path Finding* (MAPF) problem is the problem of finding a plan for a set of agents that moves the agents from their current vertices to their target vertices without collisions. MAPF has practical applications in video games, traffic control, and robotics (see Felner et al. (2017) for a survey). In many cases, there is also a requirement to minimize some cumulative cost function, such as the time spent or costs incurred until all agents

©2020 AI Access Foundation. All rights reserved.

have reached their goals (Standley, 2010; Standley & Korf, 2011; Surynek, 2010). Solving MAPF optimally is NP-Hard (Yu & LaValle, 2013b; Surynek, 2010). Nonetheless, efficient optimal algorithms exist, some are even capable of finding optimal plans for more than a hundred agents (Wagner & Choset, 2015; Boyarski, Felner, Stern, Sharon, Shimony, Bezalel, & Tolpin, 2015; Surynek, 2012; Yu & LaValle, 2013a).

When executing a MAPF plan, however, unexpected events may delay some of the agents. When such an event occurs, one may need to adjust the plan of one or more agents to avoid collisions. However, such re-planning can be very costly or even impossible in some applications, as it may require computing and communication capabilities as well as time that the agents may not have. In such cases, one may wish to generate a plan that can withstand unexpected delays, avoiding the need to replan if they occur. This is especially desirable in safety-critical settings such as air traffic control. In this work we provide a complete solution for solving MAPF problems in a way that is robust to such unexpected delays.

In the first part of this work, we explore a novel form of *robustness* for MAPF plans that we call k-robust. A k-robust plan is a plan that is robust to k delays per agent during plan execution, i.e., each agent may be delayed up to k times during plan execution and the plan would still be safe (no collisions). There are many applications in which finding a k-robust plan is desirable. For example, consider agents with an imperfect localization mechanism, where a control mechanism is employed to keep them on track. The k parameter in such cases is dictated by how far from the plan the control mechanism allows the agent to be.

We establish necessary and sufficient conditions for a plan to be k-robust, and propose several algorithms for finding optimal k-robust plans. Specifically, we show how to adapt three existing MAPF algorithms to return k-robust plans: (1) an A* (Hart, Nilsson, & Raphael, 1968) based pathfinding algorithm, (2) an algorithm based on the Conflict-Based Search (CBS) (Sharon et al., 2015) algorithm, which is a commonly used MAPF solver, and (3) an algorithm based on Picat (Zhou et al., 2015), a declarative constraint programming language shown to be effective for solving MAPF problems (Zhou, Barták, Stern, Boyarski, & Surynek, 2017). We evaluate experimentally these proposed algorithms on several standard benchmarks identifying when each solver works best.

In the second part of this work, we address the case where more than k delays occur during plan execution. To this end, we follow the framework of Ma et al. (2017) in which a plan is coupled with an *execution policy* to handle delays online, possibly modifying the original plan. We propose and analyze several classes of execution policies and prove that they are *robust*, which means that if they are used during plan execution then the agents do not collide with each other even if unexpected delays occur. Then, we compare these robust execution policies experimentally, and analyze the different tradeoffs they provide in terms of CPU time, number of required plan modifications, and total cost of the executed plan. We show also how using a k-robust plan integrates naturally in this framework, resulting in a complete and robust solution that significantly reduces the number of times that modifications to the plan are needed during execution.

Lastly, we consider cases where communication or synchronization between the agents is imperfect, e.g., agents can not communicate or synchronize every time step or even at all. We show how our execution policies can be slightly modified in order to be robust and

avoid collisions, showing the relation between the communication delay and the required degree of robustness.

This paper is organized as follows. Section 2 provides relevant background on MAPF. Section 3 defines the notion of a k-robust plan and provides necessary and sufficient conditions for finding such a plan. Section 2.1 presents two search-based algorithms for finding optimal k-robust plan, one based on A* and the other based on CBS. Section 5 shows another way to find optimal k-robust plans using a declarative constraint programming language called Picat. Section 6 discusses how to handle unexpected delays during execution, defining several robust execution policies. Section 7 discusses how to adapt these execution policies to the case where communication between the agents may be delayed. Finally, Sections 8 and 9 present relevant prior, conclusions, and future work.

Some of the material presented in this paper have been published earlier in a conference (Atzmon, Stern, Felner, Wagner, Barták, & Zhou, 2018). This paper provides a more comprehensive and rigorous treatment of k-robust MAPF, including a new theorem that specifies the relation between robust plans and conflicts (see Section 3), a more comprehensive experimental evaluation, formal treatment of execution policies (see Section 6), and a new extension of dealing with robust execution policies when facing unexpected events with delayed communication (see Section 7).

2. Background

The *Multi-Agent Path Finding* (MAPF) problem is defined by a graph, $G = (V, E)$ and a set of n agents labeled $a_1 \dots a_n$, where each agent a_i has a start vertex $s_i \in V$ and a goal vertex $g_i \in V$. Time is discretized into time steps, and in each time step an agent can perform an *action*. It can either *move* (move action) to an adjacent vertex or *wait* (wait action) in its current vertex. A solution to a MAPF problem is a plan $\pi = \{\pi_1, \dots, \pi_n\}$ such that $\forall i \in \{1, \dots, n\}$, a path π_i is a sequence of adjacent vertices (from V) leading agent a_i from s_i to g_i. We denote by $\pi_i(t)$ the expected location (vertex) of agent a_i at time t. Thus, $\pi_i(0) = s_i$ and $\pi_i(|\pi_i| - 1) = g_i$. Notice that each two consecutive vertices in π_i must have an edge between them, i.e., $\forall t : (\pi_i(t), \pi_i(t+1)) \in E$.

Definition 2.1 (Conflict)**.** A vertex conflict $\langle a_i, a_j, t\rangle$ in a plan π occurs iff agents a_i and a_j are planned to occupy the same vertex at time t, that is, if $\pi_i(t) = \pi_j(t)$. A swapping conflict $\langle a_i, a_j, t\rangle$ in a plan π occurs iff agents a_i and a_j are planned to traverse the same edge at time t, that is, if $(\pi_i(t-1) = \pi_j(t)) \wedge (\pi_i(t) = \pi_j(t-1))$. A conflict $\langle a_i, a_j, t\rangle$ in a plan π occurs iff $\langle a_i, a_j, t\rangle$ is either a vertex conflict or a swapping conflict (Stern et al., 2019).[1]

A plan π is said to have a conflict if there exists a pair of agents (a_i, a_j) and a time step t for which $\langle a_i, a_j, t\rangle$ is a conflict. We say that π is a **valid plan** if it does not have any conflict. A MAPF solver is *sound* if it outputs a valid plan. The *sum-of-costs* (SOC) of a plan π, is defined in this work as the sum of actions planned for the agents in π, i.e., $SOC(\pi) = \sum_{i=1}^{n}(|\pi_i| - 1)$. The *makespan* of a plan π is the maximal number of actions in its paths, i.e., the makespan of π is $\max_{i=1}^{n}(|\pi_i| - 1)$. The optimization criteria for MAPF

1. In this work we allow following conflicts and cycle conflicts. See discussion on these types of conflicts in Stern et al. (2019).

plans that we consider in this work is SOC. Thus, the *cost* of a MAPF plan is defined here as its SOC, and a plan is *optimal* for a MAPF problem if it has the minimal SOC among all other valid plans for that problem.

2.1 Conflict-Based Search

Conflict-based search (CBS) (Sharon et al., 2015) is a state-of-the-art MAPF solver with many extensions (Ma et al., 2017; Boyarski et al., 2015). CBS does not search the n-agent state space explicitly. Instead, it finds a plan by searching for a path for each agent separately. Conflicts are avoided by imposing a set of *constraints* of the form $\langle a_i, v, t\rangle$, representing that agent a_i is prohibited from occupying v at time step t. A plan $\pi = \{\pi_1, \dots \pi_n\}$ is called *consistent* with a set of constraints $\mathcal{C}$ if its paths satisfy all the constraints in $\mathcal{C}$, i.e., for every constraint $\langle a_i, v, t\rangle \in \mathcal{C}$, it holds that $\pi_i(t) \neq v$.[2]

CBS searches a *constraint tree* (CT) for a set of constraints such that a plan consistent with this set of constraints is valid and optimal. The CT is a binary tree, in which each node N represents:

1. A set of constraints imposed on the agents ($N.constraints$)

2. A plan ($N.\pi$) consistent with these constraints

We denote by $N.cost$ the SOC of the plan $N.\pi$.

The root of the CT contains an empty set of constraints (thus, every plan is consistent with the root). A successor of a node in the CT inherits the constraints of the parent node and adds a single new constraint for one agent. Generating a successor node N means finding a plan consistent with $N.constraints$ and identifying the conflicts in this plan, if they exist. With the exception of the root node, every node N in the CT was generated by adding a single new constraint. Therefore, in every CT node except the root, only one agent needs to replan, which can be done with any optimal single-agent path-finding algorithm, e.g., A*. The algorithm used for this purpose is referred to as the CBS low-level solver. A CT node N is a goal node when $N.\pi$ is valid. To search the CT for a goal node CBS runs a best-first search where nodes are ordered by their costs ($N.cost$).

The two key components of CBS are how to identify conflicts in a CT node N, and how to choose which constraint to add when expanding N and generating its successors. We describe them below.

Identifying conflicts in a consistent plan. Once a consistent plan has been found by the low-level solver, it is *validated* by simulating the movement of the agents along their planned paths ($N.\pi$) and searching for conflicts between the agents. That is, for every time step $t = 0$ up to the makespan of $N.\pi$, and for every pair of agents a_i and a_j, CBS checks if $\langle a_i, a_j, t\rangle$ is a conflict (vertex or swapping). If all agents reach their goals without any conflict, N is declared as a goal node, and $N.\pi$ is returned. Otherwise, a conflict is found and the node is declared a non-goal.

2. In CBS, it is also possible to impose a constraint of the form $\langle a_i, e, t\rangle$, representing that agent a_i is prohibited from traversing the edge e at time step t. For a plan π to be consistent with a set of constraints $\mathcal{C}$, it must also satisfy all constraints in $\mathcal{C}$, including these edge constraints. That is, for every constraint $\langle a_i, e, t\rangle$ where e is an edge from v to v', it holds that $(\pi_i(t) \neq v) \vee (\pi_i(t+1) \neq v')$.

Resolving a conflict (expanding CT nodes). When a non-goal CT node N is chosen in the best-first search of the CT, CBS generates its successor CT nodes as follows. First a conflict $\langle a_i, a_j, t\rangle$ is chosen from one of the conflicts in $N.\pi$. Note that since N is not a goal, it contains at least one conflict. Let v be the vertex of this conflict, i.e., $v = N.\pi_i(t) = N.\pi_j(t)$. CBS *expands* N and generates two new CT nodes, adding the constraint $\langle a_i, v, t\rangle$ to one child and the constraint $\langle a_j, v, t\rangle$ to the other child. To obtain a consistent plan for each child CT, the low-level search is activated to the agent for which the new constraint was added. All other agents maintain their paths.

2.2 Robust Execution Policies

Ma et al. (2017) also considered the case where unexpected delays occur while the agents are executing a MAPF plan. For such cases, they suggested a robust execution policy called *Minimal Communication Policy* (MCP). During execution, MCP preserves the order by which agents visit each vertex as in the original plan. That is, when an agent i is about to perform a move action and enter a vertex v, MCP checks whether agent i this agent is the next agent to enter that vertex by the original plan. If a different agent, j, is planned to enter v next according to the original plan, then i waits in its current vertex until until j leaves v. In Section 6 we suggest several additional execution policies and compare them to MCP. Moreover, we extend these execution policies in such a way that would still prevent the collision of agents even in settings where the agents cannot communicate after each time step.

3. k-Robust MAPF

In this section, we define the notion of a k-robust plan, provide necessary and sufficient conditions for finding a k-robust plan, and defining what is an optimal k-robust plan.

A *delay* in an execution of a plan π is defined by a tuple $\langle a_i, t\rangle$, representing that agent a_i at time t stayed in the vertex in which it was at time $t-1$ instead of performing the action planned for it and moving to $\pi_i(t)$. After experiencing a delay, an agent can try to continue to follow the plan. As a result, the agents may end up executing a plan that is different from the original plan. Formally, the planned execution of a plan π after experiencing delay $D = \langle a_i, t_D\rangle$, denoted $D[\pi] = \{D[\pi_1], \ldots, D[\pi_n]\}$, is the MAPF plan defined as follows:

$$D[\pi_j](t) = \begin{cases} \pi_j(t) & j \neq i \\ \pi_i(t) & j = i \wedge t < t_D \\ \pi_i(t-1) & otherwise \end{cases} . \tag{1}$$

The planned execution of π after experiencing a set of delays $\mathcal{D} = D_1, \ldots D_T$, denoted $\mathcal{D}[\pi]$, is defined as $D_T[D_{T-1}[\cdots D_1[\pi]]\cdots]$. A plan π is *robust* to a delay D if the delayed agent can continue to follow its path after the delay without causing a collision, i.e., if $D[\pi]$ is a valid plan. π is robust to a set of delays $\mathcal{D}$ iff experiencing any subset of $\mathcal{D}$ yields a valid plan. A plan is k-robust iff it is valid, and it is robust to any set of delays that contains at most k delays for each agent.

As an example of how delays affects the execution of a MAPF plan. Figure 1(a) shows a valid plan π with two agents a_1 and a_2 with paths $\pi_1 = (s_1, C, g_1)$ and $\pi_2 = (s_2, A, B, C, g_2)$,

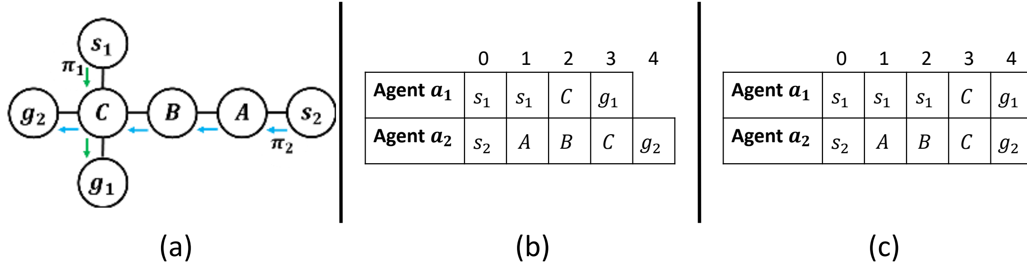

Figure 1: (a) A MAPF plan π (b) The planned execution of π at time 1, after agent a_1 experienced a delay (c) The planned execution of π at time 2, after agent a_1 experienced 2 delays.

respectively. Figure 1(b) presents the execution of π after agent a_1 experienced a delay at time 1. This delay is defined by $D = \langle a_1, 1\rangle$ and the planned execution of π after experiencing this delay is $D[\pi] = \{D[\pi_1], D[\pi_2]\}$ where $D[\pi_1](1) = (s_1, s_1, C, g_1)$ and $D[\pi_2](1) = (s_2, A, B, C, g_2)$. Note that $D[\pi]$ is still a valid plan. Therefore, we can say that π is a robust plan w.r.t. D. Figure 1(c) presents the execution of π after agent a_1 experienced two delays, at time 1 and at time 2. Formally, these delays are defined by $D_1 = \langle a_1, 1\rangle$ and $D_2 = \langle a_1, 2\rangle$, and the planned execution after experiencing these delays is $D_2[D_1[\pi]] = \{D_2[D_1[\pi_1]], D_2[D_1[\pi_2]]\}$ where $D_2[D_1[\pi_1]] = (s_1, s_1, s_1, C, g_1)$ and $D_2[D_1[\pi_2]] = (s_2, A, B, C, g_2)$. Note that $D_2[D_1[\pi]]$ is not a valid plan because both agents are located in C at the same time, hence, π is not a 2-robust plan.

Definition 3.1 (The kR-MAPF problem)**.** A kR-MAPF problem is defined by a MAPF problem and a non-negative integer value k. A solution to a kR-MAPF problem is a k-robust plan to the given MAPF problem.

3.1 The Relation Between Robust Plans and Conflicts

A brute-force approach to check if a plan π is k-robust is to check every set of $k' \leq k$ conflicts, compute the planned execution of π after experiencing these conflicts, and checking if it is valid. Next, we provide a simpler and more efficient way to check if π is k-robust.

Definition 3.2 (k-delay Conflict)**.** A k-delay conflict $\langle a_i, a_j, t\rangle$ in a plan π occurs iff there exists $\Delta \in \{0, \ldots, k\}$ such that agents a_i and a_j plan to occupy the same vertex in timesteps t and $t + \Delta$, respectively, i.e, $\pi_i(t) = \pi_j(t + \Delta)$.

Checking if a plan π has a k-delay conflict can be done in time that is polynomial in k, n (the number of agents), and the makespan of π.

Observation 1. A plan is k-robust iff it does not contain any k-delay conflicts.

We provide a formal proof of Observation 1 in Appendix A. Observe that swapping conflicts cannot exist in a k-robust plan for any $k > 0$. [3] Also, k-robust plans with $k > 0$ do not allow agents to move to a vertex occupied by another agent in the previous time

3. If a swapping conflict $\langle a_i, a_j, t\rangle$ exists in a plan π then by definition $\langle a_i, a_j, t\rangle$ is also a 1-delay conflict in π (see Definitions 2.1 and 3.2).

step. Such a train-like motion (lockstep motion) is not allowed in some MAPF formulation (Kornhauser, Miller, & Spirakis, 1984). Following the terminology defined by Stern et al. (2019), we do not allow vertex, edge, or swapping conflicts in the solution, but do allow following and cycle conflicts.

3.2 Optimal kR-MAPF

More than one k-robust plan may exist for a given MAPF problem. Since plans with lower *costs* are preferred, we say that a k-robust plan is **optimal** if there is no other k-robust plan with a lower cost.

Lemma 3.1. For any k_1 and k_2 where $k_1 < k_2$ it holds that the cost of the optimal k_1-robust plan is *smaller than or equal to* the cost of the optimal k_2-robust plan.

Proof. A plan π' is k_1-robust if there are no k_1-delay conflicts (Observation 1). Since $k_1 < k_2$, every plan π'' that is k_2-robust is also a k_1-robust plan. Thus, the set of all k_2-robust plans is a subset of the set of all k_1-robust plan. Hence, the cost of an optimal k_1-robust plan is *smaller or equal to* the cost of an optimal k_2-robust plan. □

As a consequence there is a tradeoff between the robustness of a plan and its cost. In this work, we consider the robustness k as a hard constraint and optimize the plan cost. That is, we explore several algorithms for finding an optimal k-robust plan, for a given value of k.

4. Search-Based Solutions

In this section, we propose two algorithms for finding optimal k-robust plans, that are based on graph search techniques.

4.1 A*

Several known MAPF algorithms (Silver, 2005; Standley, 2010; Goldenberg, Felner, Stern, & Schaeffer, 2012; Wagner & Choset, 2015) are based on the well-known A* algorithm (Hart et al., 1968). These algorithms search in a state space called the *n-agent state space*. A *state* in this state space represents a possible way to place n agents into $|V|$ vertices, one agent per vertex. An *action* in this state space represents n single-agent move/wait actions, one per agent. An action is *applicable* if its constituent single-agent actions do not create a conflict. Hence, a path in this n-agent state space from the state $(s_1, \dots s_n)$ to the state $(g_1, \dots g_n)$ corresponds to a valid plan.

One way to modify A*-based solvers to return k-robust plans is to modify the way an action in this state space is defined, such that combinations of single-agent actions that lead to k-delay conflicts are prohibited. This modification by itself, however, may lead to non-optimal plans. For example, consider finding a 2-robust plan for the problem depicted in Figure 2(a). The optimal 2-robust plan is $\pi_1 = (s_1, A, B, C, g_1)$ and $\pi_2 = (s_2, D, g_2)$, with a cost of 6. Consider running A* on this problem. First, A* expands the state (s_1, s_2), generating two children (A, C) and (A, D). Assume that (A, C) was expanded first, generating state (B, g_2) with cost 4 (2 per agent). Next, (B, g_2) is expanded. Since a_2 was

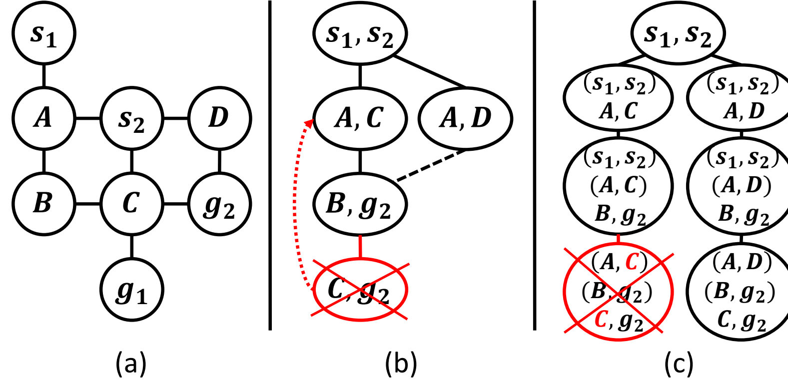

Figure 2: A MAPF problem (a) its search tree (b), and its k-robust search tree (c). The red lines show k-delay conflicts.

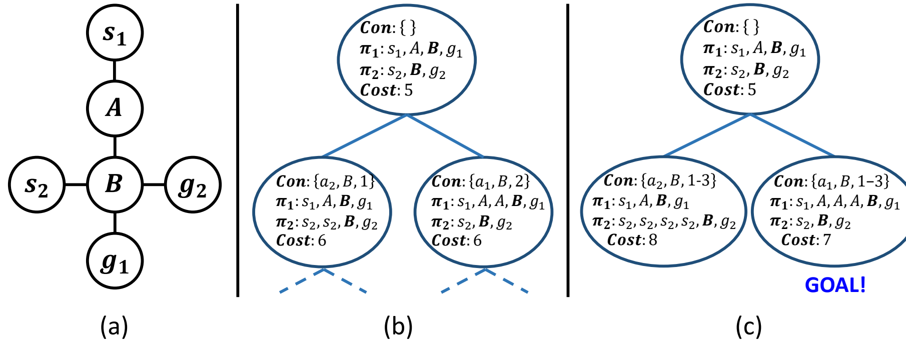

Figure 3: (a) The graph (b) The CT using the original time/vertex constraints (c) CT using the range constraints

in C at $t = 1$, state (C, g_2) will not be generated due to the 2-robustness constraint. Next, state (A, D) is expanded. It will not generate (B, g_2), as this state was already reached via state (A, C) with the same cost (see Figure 2(b)). Thus, while there is a plan in which state (B, g_2) generates state (C, g_2), this specific run of A* will not find it. As a result, A* will return a suboptimal plan of cost 7.

To remedy this, the n-agent state space needs to be modified to keep track of the last k steps of each agent in each state. In this state space, a state represents a possible way to place n agents into V vertices (out of the $\binom{|V|}{n}$ possibilities), one agent per vertex, *over k consecutive time steps.* Figure 2(c) shows the search tree of this extended state space. An A* search over this state space will return an optimal k-robust plan. However, the size of this search space grows exponentially with k. Thus, its size is much larger than the size of the n-agent state space, resulting in poor search performance.

4.2 k-Robust CBS

Next, we introduce *k-robust CBS* (kR-CBS), an adaptation of CBS designed to return optimal k-robust plans. [4] kR-CBS differs from CBS in how it identifies and resolves conflicts.

Identifying conflicts. Based on Observation 1, in kR-CBS a CT node N is a goal iff it has no k-delay conflicts. Therefore, once a consistent path is returned by the low-level solver, kR-CBS checks if the plan contains k-delay conflicts. Here too, this is done by simulating the movement of the agents along their planned paths and searching for k-delay conflicts. This is easy to implement and runs in time that is polynomial in k, the plan's makespan, and the number of agents. However, since checking if there is a k-delay conflict at time t requires checking the previous k steps, the runtime of identifying conflicts in kR-CBS is larger by a factor of k than the equivalent plan validation step in CBS.

Resolving conflicts (expanding CT nodes).

Let N be a non-goal node in a CT selected to be expanded next by kR-CBS, and let $\langle a_i, a_j, t\rangle$ be a k-delay conflict in N. This means that there is a vertex v and a value $\Delta \in \{0, \ldots, k\}$ such that $v = N.\pi_i(t) = N.\pi_j(t + \Delta)$. There is no k-robust plan in which a_i is at v at time t while a_j is at v at time $t + \Delta$. Therefore, at least one of the constraints, $\langle a_i, v, t\rangle$ or $\langle a_j, v, t + \Delta\rangle$, must be added to the CT and must be satisfied by the low-level solvers. Consequently, kR-CBS generates two children to N, each having one of these constraints.

kR-CBS is sound, because it only halts when generating a CT node that has no k-delay conflicts. A complete algorithm guarantees that if a solution exists then the algorithm will find it. kR-CBS is complete because each time it splits a CT node, a valid plan that satisfies the constraints of that CT node must also satisfy the constraints of at least one of its children. This is because a plan that satisfies the constraints of a CT node and does not satisfies the constraints of both its children cannot be a valid plan as it must contain the conflict that CT node resolves. Similarly, kR-CBS returns optimal plans, as it searches the CT in a best-first order according to the nodes' costs, and the cost of a node N is a lower bound on the cost of any valid plan consistent with $N.constraints$.

Example. Consider a 2-robust MAPF problem on the graph in Figure 3(a), with two agents whose start-goal pairs are s_1-g_1 and s_2-g_2. Figure 3(b) shows the first two levels of the CT generated by kR-CBS, where every node N shows $N.constraints$ (labeled Con), $N.\pi_1$, $N.\pi_2$, and $N.cost$.

The plan in the root is valid, but it is not 2-robust since it has a 2-delay conflict $\langle a_2, a_1, 1\rangle$ at vertex B for $\Delta = 1$ (since $\pi_2(1) = \pi_1(2) = B$). To try to resolve this conflict, kR-CBS adds the constraint $\langle a_2, B, 1\rangle$ to the left child and the constraint $\langle a_1, B, 2\rangle$ to the right child. Both children of the root node are also not goal nodes. In fact, in this example we will need to generate a total of 7 CT nodes before finding an optimal plan. As we show next, it is possible to improve kR-CBS such that it will find a goal sooner.

4.3 Improved k-Robust CBS

In Figure 3(b), the 2-delay conflict $\langle a_2, a_1, 1\rangle$ in the root CT node is resolved by adding the constraint $\langle a_2, B, 1\rangle$ to its left child and adding the constraint $\langle a_1, B, 2\rangle$ to its right child.

4. Our implementation of kR-CBS is available at `https://git.hub.com/doratzmon/k_robust`.

Imposing these constraints is correct because in every 2-robust plan either a_1 is not at B at time 1 or a_2 is not in B at time 2. This argument can be extended: in every 2-robust plan either a_2 is not in B at times 1 and 2 or a_1 is not in B at times 2 and 3. Thus, we can impose a stricter constraint on the left child of the root CT node by adding the constraint $\langle a_2, B, 2\rangle$, and adding the constraint $\langle a_1, B, 3\rangle$ to the right child of the root CT node. Imposing more constraints per CT node may possibly reduce the size of the CT tree (but will never increase it), and consequently the overall runtime might also be reduced.

To exploit this understanding, we introduce the Improved kR-CBS (IkR-CBS). IkR-CBS works exactly like kR-CBS, except that it resolves conflicts in a CT node N by adding *range constraints* to its successors. A *range constraint* is defined by the tuple $\langle a_i, v, [t_1, t_2]\rangle$ and represents the constraint that agent a_i is prohibited from occupying vertex v from time step t_1 to time step t_2.

Ideally, we would like to construct range constraints as large as possible, to minimize the size of the CT tree. However, over-constraining CT nodes may result in losing completeness and optimality. The key question in implementing IkR-CBS is which pairs of range constraints can be use to resolve conflicts without losing completeness and optimality.

Definition 4.1 (Sound Range Constraints)**.** A pair of range constraints for a given k-delay conflict is called *sound* iff every k-robust plan satisfies at least one of the constraints.

Proposition 4.1. If IkR-CBS resolves conflicts with sound pairs of range constraints then it is guaranteed to return an optimal k-robust plan if such exists.

Proof. A goal CT node IkR-CBS is a CT node that does not have any k-delay conflicts. Thus, every plan returned by IkR-CBS is a k-robust plan (Observation 1). Let $\pi(N)$ denote all the k-robust plans that satisfy *N.constraints*, and let N_1 and N_2 be the children of N, generated by a sound pair of range constraints R_1 and R_2, respectively. Observe that $\pi(N_1)$ contains all the plans in $\pi(N)$ that satisfy R_1, and similarly $\pi(N_2)$ contains all the plans in $\pi(N)$ that satisfy R_2. Since R_1 and R_2 are a sound pair of constraints, it holds that $\pi(N) = \pi(N_1) \cup \pi(N_2)$. Thus, splitting CT nodes by resolving conflicts with a sound pair of constraints does not lose any plans. Since IkR-CBS, just like kR-CBS, searches the CT in a best-first manner according to the desired optimality criteria (SOC), we have that IkR-CBS is guaranteed to find an optimal k-robust plan, if such exists. □

To implement IkR-CBS, one needs a method for finding a sound pair of range constraints. We propose a spectrum of such methods below where an important member of this spectrum is the pair of *symmetric range constraints* also defined below.

4.3.1 Finding a Sound Pair of Range Constraints

We call a pair of range constraints *symmetric* if the vertex they constrain and the corresponding time range is identical for both agents. For any time step t, vertex v, and agents a_i and a_j, a symmetric range constriant is in the form $\langle a_i, v, [t, t+k]\rangle$, $\langle a_j, v, [t, t+k]\rangle$.

Corollary 4.2 (Symmetric range constraints)**.** For any time step t, vertex v, and agents a_i and a_j, the range constraints $\langle a_i, v, [t, t+k]\rangle$, $\langle a_j, v, [t, t+k]\rangle$ are sound for solving a k-robust MAPF problem.

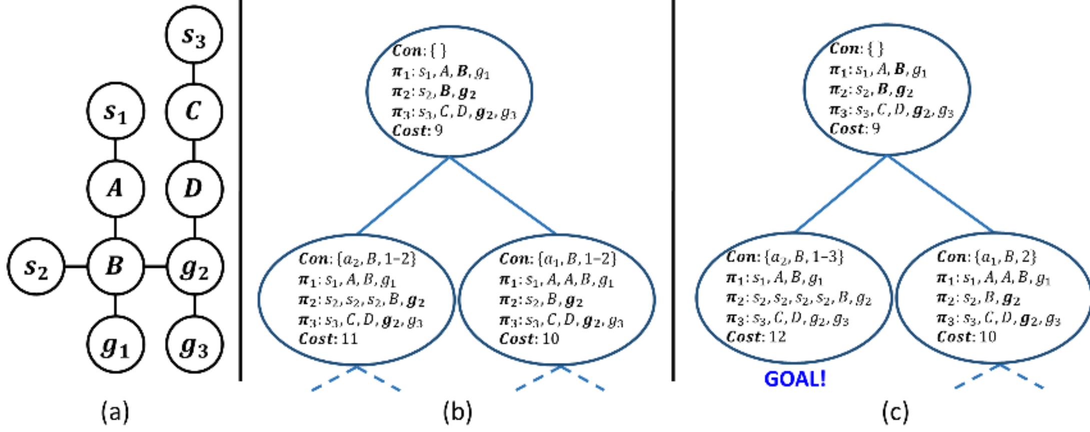

Figure 4: (a) The graph (b) The CT using the symmetric range constraints (c) CT using the asymmetric range constraints

Proving Corollary 4.2 is straightforward. Note that setting a symmetric range constraint on a range larger than k is, in general, not sound. Thus, Corollary 4.2 gives an upper bound on the size of the largest pair of symmetric range constraints that is sound. Nevertheless, there are more than one k-sized symmetric pairs of range constraints for a given k-delay conflict $\langle a_i, a_j, t\rangle$ over vertex v. In our implementation, we used the time range $[t, t+k]$ (where t is the time that the first agent arrived at v), but the pair of range constraints $[t+\Delta-k, t+\Delta]$ (where $t+\Delta$ is the time that the second agent arrived at v) is also sound.

A pair of sound range constraints can also be *asymmetric*, i.e., constrain one agent to a longer time range than the other agent. This provides a spectrum of ways to set the ranges for each agent. For example, consider a conflict $\langle a_i, a_j, t\rangle$ over vertex v and pair of range constraints $R_1 = \langle a_i, v, [t-k, t+k]\rangle$ and $R_2 = \langle a_j, v, [t]\rangle$. R_1 and R_2 are a sound pair of constraints, because a solution must satisfy either R_1 or R_2, since violating both results in a k-delay conflict. R_1 and R_2 are extremely asymmetric in the sense that one agent is restricted from occupying v in a large time range of $[t-k, t+k]$ and the other agent is restricted from occupying v only in a one single time step t. However, one can imagine a larger spectrum of asymmetric range constraints that are more balanced. An open question for asymmetric range constraints is how to choose the agent which will get the more restrictive constraint.

Figure 4 presents an example where there is an advantage of using asymmetric range constraints over symmetric range constraints. Figure 4(a) shows a kR-MAPF problem with $k=1$ where three agents a_1, a_2, and a_3 should move from their start vertices s_1, s_2, and s_3 to their goal vertices g_1, g_2, and g_3, respectively. Figures 4(b) and 4(c) show the CT of kR-CBS using symmetric range constraints and asymmetric range constraints, respectively. In both figures, the root contains the shortest paths for all three agents. However, splitting the conflict $\langle a_2, a_1, 1\rangle$ (in vertex B) results in different children. While the right child of both calculates the same paths with a different constraint, the left child of the asymmetric range constraint finds a conflict free solution and the left child of the symmetric constraints finds a non-goal plan. Thus, in Figure 4(b) the left child needs to be further expanded in order to find the solution. Overall, in this example, using symmetric range constraints

	Plan cost			Plan time (ms)						
				k=0	k=1			k=2		
n	k=0	k=1	k=2	All	KR	KR(A)	KR(S)	KR	KR(A)	KR(S)
4	22	22	22	6	17	**15**	16	229	130	**79**
6	31	31	32	5	25	26	**21**	1,087	416	**97**
8	40	41	43	7	29	24	**21**	2,535	1,007	**219**
10	49	51	54	42	166	126	**80**	22,986	7,261	**895**

Table 1: Average plan cost (over all 50 instances) and planning runtime for different CBS-based k-robust solvers, on an 8x8 open 4-neighborhood grid.

	Plan cost			Plan time (ms)						
				k=0	k=1			k=2		
n	k=0	k=1	k=2	All	KR	IKR(A)	IKR(S)	KR	IKR(A)	IKR(S)
4	176	176	177	11	**10**	13	11	**11**	14	16
6	259	259	259	13	**17**	**17**	**17**	23	**22**	**22**
8	338	339	339	44	56	55	**54**	70	70	**69**
10	413	413	414	53	710	107	**68**	740	130	**81**

Table 2: Average plan cost and planning runtime for different CBS-based k-robust solvers, on an 64x64 open 4-neighborhood grid.

results in five high-level expansions and using asymmetric range constraints results in only four expansions.

4.4 Experimental Results

Next, we compared the kR-MAPF solvers presented so far experimentally. Specifically, we compared the performance of kR-CBS, IkR-CBS using asymmetric pairs of range constraints, and IkR-CBS using symmetric pairs of range constraints. All experiments throughout this paper were executed on an Intel® Xeon E5-2660 v4 @ 2.00GHz processor with 16 GB of RAM. In all experiments we used CBS with the bypass enhancement of ICBS (Boyarski et al., 2015).

4.4.1 Open Grid

The first set of experiments are on an open 8x8 4-neighborhood grid. In each experiment, we created a random k-robust MAPF instance by randomly choosing the start and goal vertices of the agents. Then, we used the kR-MAPF solvers to obtain a k-robust plan, and we measured the total CPU runtime spent by the solvers and the cost of the obtained plan. We experimented with k=0, 1, and 2, and with 4, 6 8, and 10 agents. Note that $k = 0$ is standard CBS.

Table 1 shows the resulting plan cost and CPU runtime, averaged over 50 problems that were all solved (success rate of 100%). Each row corresponds to a different number of agents,

and the column labels KR, IKR(A), and IKR(S) present results for kR-CBS, IkR-CBS with asymmetric constraints, and IkR-CBS with symmetric constraints, respectively.

First, consider the plan costs. As can be seen, robust plans (i.e., $k > 0$) are often more costly than a plan that are not robust ($k = 0$). However, the added cost is relatively small. In fact, the largest relative increase in cost when moving from $k = 0$ to $k = 1$ was observed for 10 agents, where for $k = 0$ the average cost was 49 and for $k = 1$ the average cost increased to 51, which amounts to an increase of less than 5%.

Second, consider the runtime results. As expected, the runtime increases with k, as the constraints the agents must satisfy are stricter. For example, finding an optimal valid plan with kR-CBS for 10 agents required 42 milliseconds but finding an optimal 1-robust plan required 166 milliseconds. Both IkR-CBS variants run much faster than kR-CBS and this improvement increases when increasing k and when there are more agents, establishing that using range constraints is indeed worthwhile. When comparing the symmetric and asymmetric range constraints, we see a clear advantage for the symmetric range constraints. For example, consider finding an optimal 2-robust solution for 10 agents: it required 22,986 milliseconds for kR-CBS, 7,261 for IkR-CBS with asymmetric constraints, and only 895 milliseconds for IkR-CBS with the symmetric constraints.

To evaluate the impact of increasing the grid size, we also performed a similar experiment on open 32x32 and 64x64 4-neighborhood grids. Table 2 presents the results of this experiment on the 64x64 grid. The same general trends are observed: creating a k-robust plan takes longer runtime, the increase in cost is small, and the symmetric constraints perform better. A notable change is that there is almost no impact on solution cost of finding k-robust plans for $k = 1$ and 2, compared to the smaller 8×8 grid. This is because the 64×64 grid is less dense and thus there are more options for each agent to get to its goal. Similar trends were observed also for the 32×32 grid.

4.4.2 Dragon Age Origin Map

We also performed a similar experiment on a significantly larger map from the Dragon Age Origins (DAO) video game. Specifically, we chose the `brc202d` map, which has 43,151 vertices. This map is publicly available in the `movingai` repository (Sturtevant, 2012). We generated 50 MAPF problems by randomly placing 30 agents in this map and choosing a random goal vertex for each agent.

As observed in the experiment above, the two IkR-CBS variants were faster than kR-CBS, and among them, the IkR-CBS with the symmetric constraints was faster than the IkR-CBS with asymmetric constraints. This suggests that using symmetric ranges has a good balance in the average case and we used it in the rest of our experiments. A systematic study of the spectrum of ranges and investigating different ways to choose asymmetric ranges is left for a future study. Regarding the IkR-CBS with the symmetric constraints, out of 50 instances, 44 instances were solved for all three $k = 0$, 1, and 2. The average plan cost was 3,818.35, 3,818.43, and 3,818.53, respectively. The average runtime 213, 284, and 381 seconds, for $k = 0$, 1, and 2, respectively. These results show the same trends as observed in the open grid experiments: increasing k results in slightly higher plan costs and longer runtime. Notably, increasing k only increased plan cost slightly in these experiments, and in fact the optimal 2-robust plan often had the same cost as the optimal plan that is not robust.

	Success rate							
n	k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7
5	50	50	50	50	47	47	45	41
10	50	50	46	43	34	29	21	15
15	49	45	35	19	13	8	3	1

	Cost							
n	k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7
5	49	49	49	49	50	50	50	51
10	116	116	117	118	122			
15	163	164	166					

	Time (ms)							
n	k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7
5	5	7	7	34	58	344	1,231	9,210
10	12	1,683	36,142	53,944	93,559			
15	17	8,805	38,678					

Table 3: Number of instances solved within the allocated time out, average cost, and average planning time, on an 16x16 open 4-neighborhood grid.

This is because the map is very large and thus provides more room to find alternative plans of the same cost that are more robust.

4.4.3 Increasing k

Next, we checked the impact of increasing k to larger values beyond 2 (we tried up to 7). We created 50 random problems on a 16x16 open 4-neighborhood grid with 5, 10, and 15 agents, and solved each problem with IkR-CBS with the symmetric constraints.

Table 3 shows the number of problems solved within a timeout of 5 minutes (top frame), the average cost (middle frame), and the average planning time (bottom frame) for different values of k (columns) and different number of agents n (rows). As can be seen, increasing k and increasing n both affect the success rate. However, unless both were large, the majority of the problem instances could be solved. The average cost and time reported (middle and bottom frames) are of the instances that were solved for all k values up to the value for which more than 30 instances were solved. Namely, for 5 agents this was up to $k = 7$ (41 instances), for 10 agents up to $k = 4$ (34 instances), and for 15 agents up to $k = 2$ (35 instances). As in previous experiments, increasing k and increasing n both increase the cost and time. Remarkably, for all k values that we were able to solve, the increase to the cost was very small.

4.4.4 Warehouse Domain

We also experimented on the warehouse domain described by Ma et al. (2017) (presented in Figure 5) with kR-CBS and symmetric constraints . Table 4 shows the results for 5, 10,

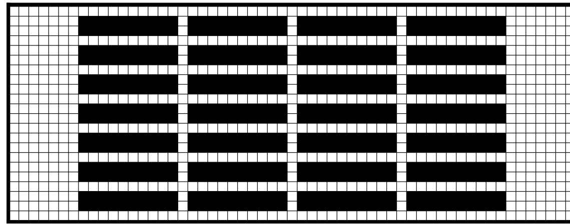

Figure 5: Warehouse domain, taken from Ma et al. (2017).

	Cost			Time (ms)		
#Agents	k=0	k=1	k=2	k=0	k=1	k=2
5	148.43	148.54	148.71	110	110	116
10	297.33	297.80	298.58	1,042	1,176	1,570
15	435.76	436.80	438.58	1,816	2,058	12,746

Table 4: Average cost and planning runtime of kR-CBS on the warehouse domain.

and 15 agents, and $k = 0$, $k = 1$, and $k = 2$. As seen in other experiments, increasing the value of k or the number of agents increase also the cost and planning time.

5. A Declarative Solution

An alternative approach to solve a MAPF problems is to compile it into another known NP-hard problem that has mature and effective general-purpose solvers. Surynek (2012, 2016) showed how MAPF problems can be solved by a SAT solver. Yu and LaValle (2013a) showed how to solve a MAPF problem optimally by formulating it as a network flow problem, which is then compiled to an integer-linear program (ILP) and solved with a ILP solver. Erdem et al. (2013) compiled MAPF into Answer Set Programming (ASP) and used an ASP solver to obtain a solution. Gange et al. (2019) replaced the high-level solver of CBS with a lazily constructed constraint programming model with nogoods (Lazy CBS). Lam et al. (2019) solved MAPF using a decomposition framework developed for mathematical optimization called branch-and-cut-and-price (BCP).

5.1 kR-MAPF using PICAT

Adapting such compilation-based solvers to return k-robust solutions is relatively simple. To demonstrate this, we implemented a MAPF solver using Picat (Zhou et al., 2015), a logic-based programming language that is publicly available. Our encoding is based on Surynek's SAT-based MAPF solver (Surynek et al., 2016), in which there is a Boolean variable for every triplet (a, t, v) of agent (a), time (t), and vertex (v), where this variable is true iff agent a occupies vertex v at time t. A set of constraints is imposed on these variables, namely:

1. Each agent occupies exactly one vertex at each time step.

2. No two agents occupy the same vertex at any time.

Obstacles	Cost		Planning time (ms)			
	k=1	k=2	k=1		k=2	
			CBS	Picat	CBS	Picat
12	35.16	36.30	1,511	**1,026**	**454**	1,805
16	39.24	40.68	4,142	**1,401**	6,572	**2,590**
19	39.67	42.47	3,362	**1,506**	8,727	**3,235**
22	34.20	36.91	6,979	**1,439**	14,042	**3,104**
25	28.26	29.52	9,834	**1,206**	12,957	**1,813**
32	16.28	18.73	**95**	322	1,488	**810**

Table 5: Solution cost and runtime results for 8x8 4-neighborhood grids with 6 agents, and a different number of obstacles.

3. In every time step an agent may only transition between two adjacent vertices.

For finding k-robust solutions, the second constraint is extended such that no two agents occupy the same vertex in time steps that are closer than k from each other. The exact Picat model we developed is available at `https://tinyurl.com/kRobust`. The advantage of using Picat to encode MAPF is that the model can be compiled to SAT, to a constraint program (CP), or to a Mixed Integer Linear Program (MILP), and then solved with an appropriate solver. In our experiments we only run a SAT compilation of our Picat model.

5.2 Experimental Results

Next, we experimentally evaluated our Picat-based solver, and compared it with IkR-CBS with symmetric range constraints, which is the CBS-based solver that performed best in our experiments.

Table 5 shows the average plan cost and CPU runtime for 50 problem instances on a 8x8 4-neighborhood grid with 6 agents and $k = 1$ and $k = 2$. The success rate is 100% for both solvers and is not reported. We experimented with problem instances with different number of randomly allocated obstacles (the “Obstacles” column). For each value of k and number of obstacles, we highlighted in bold the runtime results for the faster algorithm.[5]

As expected, for the Picat-based solver as in the CBS-based solver, increasing k results in plans of higher cost and higher runtime. For example, with 12 random obstacles, the plan cost is 35.16 and 36.30, and the average runtime for our Picat-based solver was 1,026 and 1,805, for $k = 1$ and $k = 2$, respectively.

For both algorithms, the impact of varying the number of obstacles on the planning time follows a classical easy-hard-easy pattern: with either a few or many obstacles is easy, and it becomes harder for the middle-ground, where the problem is not under- or over-constrained. For example, for $k = 1$ our Picat-based solver required an average runtime of 1,026 ms for 12 obstacles, going up to 1,506 ms for 19 obstacles, and going down to 322 ms for 32 obstacles.

Now we compare the results of kR-CBS and the Picat-based solver. Since both solvers return optimal solutions, their solution cost is the same, and the relevant comparison be-

5. The process of creating problem instances was done by, first, randomly placing obstacles. Then, setting agents such that the instances remain solvable.

#Agents		10	15	20	25	30	35	40	45	50
k=1	Picat	**50**	**50**	**47**	**35**	15	2	0	0	0
	CBS	**50**	49	42	27	**22**	**7**	**1**	0	0
k=2	Picat	**50**	**49**	**38**	**9**	0	0	0	0	0
	CBS	**50**	45	31	6	**4**	0	0	0	0

Table 6: Number of instances solved within the allocated time out. The grids used are 32x32 4-neighborhood grids with 20% obstacles.

		Cost			Time		
#Agents		10	15	20	10	15	20
k=1	Picat	228.2	337.9	438.9	63.0	97.6	**111.3**
	CBS	228.2	337.9	438.9	**15.1**	**33.3**	249.8
k=2	Picat	228.9	339.7	441.6	88.6	173.8	**193.7**
	CBS	228.9	339.7	441.6	**15.3**	**107.4**	1,455.8

Table 7: Planning cost and planning time. The grids used are 32x32 grids with 20% obstacles.

tween them is only in their runtime.For this small 8x8 grid, the Picat-based solver shows better performance in most settings. For example, consider the results in Table 5 for 25% randomly allocated obstacles. The average runtime results for $k = 1$ and $k = 2$, were 1,206 and 1,813 ms for the Picat-based solver and 9,834 and 12,957 ms for the CBS-based solver, respectively. Indeed, compilation-based approaches are known to perform well for small and relatively dense grids (Surynek et al., 2016).

Next, we compared the Picat-based and CBS-based solvers on a larger, 32x32 grids, with 20% obstacles at random cells, with 10, 15, 20, 25, and 30 agents, and $k = 1$ and $k = 2$. These problems were harder to solve and thus we set a 5-minutes timeout for every problem instance. Table 6 shows the number of instances solved under this timeout, out of a total of 50 problem instances. Here, the results of both solvers are very similar, and there is no clear advantage to either. Table 7 presents the average cost and time of the same experiment, of the instances that both solvers were able to solve for both values of k. In our case, this amounted to 50, 44, and 31 instances for 10, 15, and 20 agents, respectively. As can be seen, in terms of planning time, the CBS-based solver is more sensitive to increasing k and n. For example, for $k = 2$ solving for 20 agents is almost 100 times slower than when solving for 10 agents. In contrast, solving for 20 agents with Picat is less than 3 times slower compared to 10 agents.

Finally, we experimented with the `brc202d` DAO map described earlier (we used the same instances as in Section 4), which is much larger than the 32x32 4-neighborhood grid used in Table 6. Here, the Picat-based solver was not able to solve any problem instance, even with only 5 agents. By contrast, the CBS-based solver was able to find even optimal 2-robust solutions for some instances with 95 agents. Figure 6 shows the number of problems

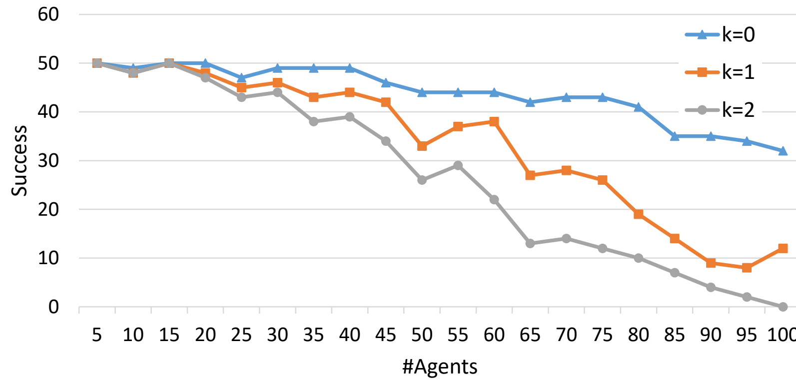

Figure 6: Success rate for kR-CBS over the `brc202d` DAO map.

solved within the 5-minutes timeout from a total of 50 problem instances, with $k = 0, 1$, and 2 and 5, 10, 15, .. ,100 agents.

In conclusion, there is no universal winner: for small grids the Picat-based solver is best, while for very large grids the CBS-based solver is much better. This trend was also observed in prior works in regular MAPF (Surynek et al., 2016; Zhou et al., 2017).

6. Robust Execution

A k-robust plan is not enough to provide a completely robust solution, since the parameter k is chosen a-priory while during plan execution more than k delays may occur. To address this challenge and provide a holistic robust solution, we propose to use the framework introduced by Ma et al. (2017) and mentioned in Section 2.2. In this framework, delays are handled online according to a given *execution policy*. An execution policy specifies if and how to modify π to take into account the experienced delays.

In this section, we propose several intuitive execution policies that integrate well with having a k-robust plan, and under certain condition guarantee that a plan can be safely executed even in the presence of unexpected delays. As a preliminary, we formally define the notion of a *robust* execution policy and discuss how it relates to prior work by Ma et al. (2017).

6.1 Robust Execution Policies

We say that an execution policy is *robust with respect to a plan* π iff it prevents all collisions during the execution of π. Our definition of robust execution policy generalizes the concept of *robust plan-execution policy* introduced by Ma et al (2017). They defined such an execution policy as a policy that prevent all collisions during execution of a *valid MAPF-DP plan.* A valid MAPF-DP plan is a plan that has the following two properties:

- $\forall i, j, t$ with $i \neq j : \pi_i(t) \neq \pi_j(t)$ [two agents can not be in the same vertex at the same time]

- $\forall i, j, t$ with $i \neq j : \pi_i(t+1) \neq \pi_j(t)$ [an agent cannot be in time step $t+1$ in the same vertex that another agent was at time step t (i.e., no following conflicts (Stern et al., 2019))]

Ma et al. (Ma et al., 2017) proved that the MCP execution policy described in Section 2.2 is a robust execution policy.

Observe that a MAPF-DP plan is exactly a 1-robust plan (Definition 3.1). Since for every $k > 1$, every plan that is k-robust is also 1-robust, this means that MCP is a robust execution policy w.r.t any plan that is k-robust for $k \geq 1$. However, there are other execution policies that are also robust w.r.t to such plans. Next, we describe several classes of such execution policies.

6.2 When to Modify the Plan

Here, we define three classes of execution policies. They differ in *when* they choose to modify π:
(1) Eager. Modify π immediately when a delay occurs.
(2) Reasonable. Modify π when a delay occurs but only if that delay is expected to cause a conflict later between the delayed agent and some other agent.
(3) Lazy. Modify π only when a conflict is expected to occur in the next time step.

Checking whether a conflict is expected to occur is done by simulating the execution of π from the current vertices of the agents. The logic behind “Reasonable” is that if a conflict is expected to occur then it is best to modify π to avoid it as soon as possible, while the logic behind “Lazy” is that future unexpected delays may resolve expected conflicts even without modifying π earlier.

6.3 How to Modify a Plan

Execution polices can also be classified by *how* the plans are modified:

- **Replan.** Modify π by creating a completely new plan starting from the current time step using a MAPF solver.[6]

- **Repair.** Modify π by performing minor modifications to it.

In general, there is a tradeoff between replan or repair policies. Replan policies incur significant CPU overhead compared to repair policies, but they may end up having lower execution cost. In particular, when there are many delays it may be better to create a completely new plan than to modify the original one.

There may be many ways to repair a plan (Felner, Stern, Rosenschein, & Pomeransky, 2007), but in this work we focus on repair execution policies that do so by forcing some of the agents that were not delayed to wait. We distinguish between two types of such repair policies:

- **All.** All agents that were not delayed in the current time step are forced to wait at the next time step.

6. In our experiments, we used the same MAPF solver used to create the original plan π, but other MAPF slover can also be used.

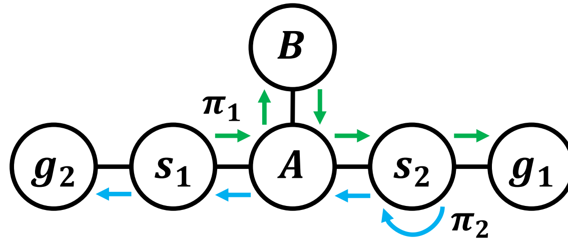

Figure 7: Example where lazy repair leads to a deadlock

- **Selective.** Some of the agents that were not delayed in the current time step are forced to wait at the next time step.

The advantage of an All repair execution policy is that unless other delays occur, the configuration at the next time step is identical to the configuration planned for the current time step. This is advisable in scenarios where preserving the relative positions of agents during the plan is important. The advantage of a Selective repair execution policy is that it may save execution cost, as it enforces fewer agents to wait. MCP (Ma et al., 2017) is an example of a selective repair execution policy.

6.4 Combinations of Execution Policies

The choice of when to modify π is orthogonal to the choice of how to update it. Thus, we can define 9 classes of execution policies: {Eager, Reasonable, Lazy} $\times$ {Replan, Selective repair, All repair}. However, all combinations of Lazy and repair (either Selective or All) are ill-defined since repair execution policies consider the agents that were not delayed in the current time step, but Lazy execution policies are only invoked before a conflict is about to occur. Thus, it may be the case that a lazy policy is invoked in a time step in which no agent has been delayed at all, as the delays occurred several time steps earlier.

Moreover, a lazy repair policy may end up in a deadlock, as demonstrated in Figure 7. Assume that the original 1-robust plan π is: $\pi_1 = (s_1, A, B, B, B, A, s_2, g_1)$ and $\pi_2 = (s_2, s_2, s_2, A, s_1, g_2)$. If a_1 is delayed in s_1 for two time steps, we reach a state where a_1 is at s_1 and a_2 is at A. At this stage, a conflict occurs as according to π the agents will cross paths. But, enforcing either of them to wait will not help. Thus, we did not implement any Lazy repair execution policy.

Corollary 6.1. All replan execution policies are robust w.r.t. any k-robust plan for any $k > 0$ if it replans using a planner that generates 1-robust plans

Proof. Since all edges are undirected, agents can always return to their initial start state, e.g., by tracing back. Thus, every replan execution policy is robust. □

Corollary 6.2. Eager-All and Reasonable-All are robust w.r.t. any k-robust plan for any $k > 0$.

Proof. Eager-All immediately reverts back to a situation without the delay. Thus the initial plan given by the solver, which is 1-robust, can be directly activated and will not cause a collision, unless new delays occur which will be handled too.

	Cost	#Mod.	Time (ms)	Cost	#Mod.	Time (ms)	Cost	#Mod.	Time (ms)
	Delay probability = 0.001			Delay probability = 0.01			Delay probability = 0.1		
Eager All	111.08	0.26	**0**	128.33	1.99	**0**	521.24	27.17	**0**
Reasonable All	108.51	**0.06**	**0**	117.20	**0.48**	**0**	282.93	9.11	**0**
MCP	108.34	0.21	**0**	111.08	2.79	**0**	127.88	7.38	**0**
Eager replan	**108.10**	0.2	14	**108.28**	1.04	254	**111.17**	26.47	27,313
Reasonable replan	108.18	0.08	16	108.56	0.93	975	115.76	11.75	11,332
Lazy replan	108.41	0.08	3	110.22	0.87	387	125.94	**3.62**	2,471

Table 8: Comparison of the different replanning polices.

Consider now the Reasonable-All policy. Whenever an agent gets a delay that will cause a collision further in the plan (conflict), this policy fixes it by forcing the other agents to wait. Thus, the order in which the agents arrive to a vertex, for every vertex, remains as before. Therefore, Reasonable-All keeps the same agents order at vertex as in MCP, which was proven by Ma et al. (2017) to be robust w.r.t any 1-robust plan. □

The above discussion about robust execution policies assumed that agents obtain knowledge about the delays that have occurred immediately *after* they all move.[7] Other options may exist. For example, the agents might know who is delayed *before* moving. In this case, Corollaries 6.1 and 6.2 can be modified to require only 0-robust plans and planners that generate 0-robust plans. We used this assumption in our experiments.

Note that we did not analyze in the above the Eager-Selective repair and Reasonable-Selective repair execution policies. This is because these classes of execution policies represent a range of execution policies that differ by how they select which agents are forced to wait. This choice directly affects their robustness.

6.5 Experimental Results

We compared experimentally the Eager All, Reasonable All, MCP, Eager Replan, Reasonable Replan, and Lazy Replan execution policies. Delays occurred with probability p per each move of each agent, where p is a parameter. We experimented with $p = 0.1, 0.01$, and 0.001. In this set of experiments the original plan was optimal, but without any k-robust guarantee. As mentioned above, in our experiments agents know who is delayed before moving and hence it is sufficient to use a 0-robust plan. Later in the paper, we investigate cases in which we must use robust plans for $k > 0$. Table 8 presents the results of these experiments on an 8x8 open 4-neighborhood grid with 20 agents, averaged over 50 instances. Similar trends were observed for other settings. Column "Cost" reports the execution costs, i.e., the sum of costs incurred until all agents reached their goals. Column "# Mod." reports the number of times the plan was modified by an execution policy. Column "Time (ms)" reports the CPU runtime in milliseconds (ms) required by the execution policies until all agents reached their goal. We highlighted in bold the lowest cost, # modifications, and time for every value of p.

As expected, using the replan policies results in significantly lower execution costs compared to the repair polices. For example, with delay probability of 0.1, Eager Replan yielded an average execution cost of 111.17, while Eager All, Reasonable All, and MCP yielded an

7. We note that in these plans following moves do not exist as $k > 0$.

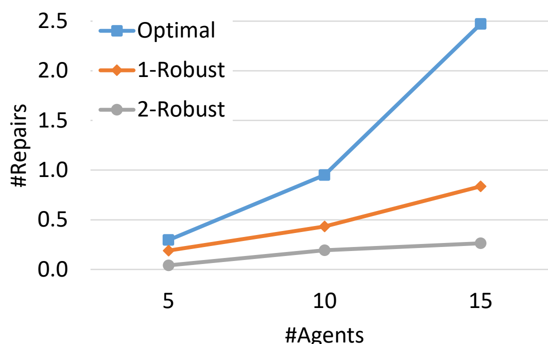

Figure 8: kR-CBS for MCP on the instances succeeded in Table 3 on a 16x16 grid

execution cost of 521.24, 282.93, and 127.88, respectively. On the other hand, repairing is done almost instantaneously while the runtime incurred when using the replan execution policies is non-negligible. This establishes a natural tradeoff between execution cost and (re)planning time.

Next, consider the relative performance of the evaluated repair execution policies, Eager All, Reasonable All, and MCP. MCP achieves the minimal execution cost. This is because it is a selective repair policy, forcing only a subset of the agents to wait instead of all of them.

Lastly, consider the impact of the delay probability parameter p. As can be seen, both execution costs and planning time increase with p. For example, the Eager All execution policy yielded an execution cost of 111.08 for $p = 0.001$ and 521.24 for $p = 0.1$. This trend is expected since higher delay probabilities means more delays and larger deviation from the original plan.

To summarize, each policy has pros and cons and one should choose the policy that is best suited for the given circumstance. An Eager replan execution policy provided the lowest execution cost, but at the cost of a high running time. All repair policies were extremely fast, and among them MCP yields the lowest solution cost by intelligently choosing which subset of agents to force to wait.

6.5.1 Using k-Robust Plans

Corollaries 6.1 and 6.2 established that all the execution policies we experimented with are robust w.r.t. k-robust plans for any $k > 0$. The next set of experiments were designed to study the influence and benefit of using k-robust plans as input to our execution polices, for a range of k values.

Figure 8 shows the results of using a k-robust plan as the original plan when using the MCP execution policy, on a 16x16 open grid. This experiment is based on the solved instances described in Table 3, executed with delay probability of 0.1. We chose MCP since it provided a good balance between execution cost, number of modifications, and replanning time (see Table 8). The x-axis shows the number of agents and the y-axis shows the average number of modifications. The different curves represent different values of k. The results show the benefit of using a k-robust plan: by increasing k we reduce the number

of modifications. Indeed, when $k = 2$ the number of modifications is close to zero, even for 15 agents, while it increases to 2.5 when using the optimal plan (where $k = 0$).

7. Delayed Communication

So far, we assumed that agents can communicate with each other instantaneously, and that they know about the delays experienced by all agents before moving. However, in many settings communication between agents takes time (Spaan, Oliehoek, & Vlassis, 2008; Qin, Gomez, & Orosz, 2017; Wu, Zilberstein, & Chen, 2009). In particular, consider the case where agents only communication to each other their status (including who got delayed) every fixed number of time steps. Let T be this fixed number. We refer to T as the *communication frequency*, and call every T^{th} time step a *communication time step*. In this setting, the agents know about each others' delays only during communication time steps, which can be up to T time steps since the delay has actually occurred. As we show below, the notion of k-robust plans is particularly useful for establishing robustness in this setting.

7.1 When to Modify the Plan

We adapt the Eager, Reasonable, and Lazy classes of execution policies described in Section 6.2 to the settings with $T > 0$ delay frequency, as follows.

- **Eager.** If an agent experiences a delay, then modify π in the next communication time step.
- **Reasonable.** If an agent experiences a delay, then in the next communication time step check if there is a T-delay conflict. If it exists, modify π.
- **Lazy.** Modify π only if there is a T-delay conflict in the next T moves. Perform this check only in communication time steps.

7.2 How to Modify the Plan

Next, we adapt the replan and repair classes of execution policies described in Section 6.3 to our settings, which has $T > 0$ delay frequency.

- T **Robust replan.** Modify π by creating a new T-robust plan starting from the current time step.
- T **Robust repair.** Modify π by performing minor modifications to π to make it T robust.

We propose two T robust repair methods called T robust all repair (T-All) and T robust selective repair (T-Selective). In T-All, some agents are forced to wait in a way that ends up with exactly the same amount of delay in executing their plans. Thus, if the original plan was T robust then the repaired plan will also be T robust. In T-Selective, just enough agents are forced to wait so as to avoid all T-delay conflicts. Formally, these plan repair methods are described as follows.

- **T-All**. Let Δ_i be the number of delays agent i experienced in the last T time steps and let Δ_{max} be the maximum over all Δ_i. In a T-All action, each agent needs to wait $\Delta_{max} - \Delta_i$ time steps.
- **T-Selective**. This plan repair method is an iterative process that continues as long as there is a T-delay conflict in the current plan π. Let $\langle a_i, a_j, t\rangle$ be such a conflict, and and let $\Delta \leq T$ be the Δ for which $\pi_i(t) = \pi_j(t+\Delta)$. The T-selective plan repair action will force agent a_j to wait $T - \Delta$ time steps in its current vertex, thereby resolving the corresponding T-delay conflict.

7.3 Robustness with Communication Delays

Next, we explore which combination of {Eager, Reasonable, Lazy } × {T-Robust, T-All, T-Selective } execution policies are robust execution plans in settings with communication frequency $T > 0$.

Theorem 7.1. *Eager T-replan, reasonable T-replan, and lazy T-replan, are robust for a given communication frequency T w.r.t to an initial plan that is T-robust.*

Proof. In each of these execution policies, in every communication time step t it is guaranteed that the executed plan has no T-delay conflict in the next T time step. Thus, no conflict will occur until the next communication time step. This continues until all agents reach their goal. □

Theorem 7.2. *Eager T-All, Reasonable T-All, and Reasonable T-Selective, are robust for a given communication frequency $T > 0$ w.r.t to an initial plan that is T-robust.*

Proof. Eager T-All and Reasonable T-All maintain that the plan being executed is always T-robust. Thus, their robustness follows the same reasoning as the proof of Theorem 7.1. For Reasonable T-Selective, an agent only moves to a potential conflict if the agents that was supposed to be in that vertex before it has already moved out of it. Thus, robustness follows. □

7.3.1 MCP and Communication Delays

As mentioned above, when using the MCP execution policy (Ma et al., 2017), agents preserve the order in the original plan in which they enter and leave each vertex. That is, if in the original plan π agent a_i and agent a_j are both planned to enter a vertex *loc* such that a_i enters *loc* before a_j does, then agent a_j will not enter *loc* until it receives a message from a_i that it has already exited *loc*. MCP is a robust execution policy thanks to this *order-preserving* mechanism.

In our delayed communication setting, agents can share their location (vertex) status only during communication time steps. Thus, following MCP may result in agents being forced to wait in their current vertex even if no delay has been experienced. Nevertheless, it is easy to see that MCP's order-preserving policy guarantees robustness even in this delayed communication setting.

	Cost				Time (ms)			
Delay probability = 0.1								
	T=0	T=2	T=4	T=6	T=0	T=2	T=4	T=6
Eager T-All	89.53	72.12	69.58	73.61	0.00	0.00	0.00	0.00
Reasonable T-All	53.86	56.63	60.43	66.48	0.00	0.00	0.00	0.00
Reasonable T-Selective	**48.72**	53.03	58.78	66.00	0.00	0.00	0.00	0.00
MCP	**48.72**	**52.31**	**58.74**	**65.80**	0.00	0.00	0.00	0.00
Eager T-replan	47.57	52.37	58.65	**62.96**	9.06	13.30	12.36	11.84
Reasonable T-replan	**47.53**	53.12	58.25	65.85	7.23	9.57	5.68	3.51
Lazy T-replan	47.57	**51.94**	**57.99**	65.74	6.07	10.09	11.06	9.64
Delay probability = 0.2								
	T=0	T=2	T=4	T=6	T=0	T=2	T=4	T=6
Eager T-All	220.18	95.56	81.94	84.92	0.00	0.00	0.00	0.00
Reasonable T-All	67.67	68.22	70.33	74.59	0.00	0.00	0.00	0.00
Reasonable T-Selective	**55.64**	60.55	66.01	72.76	0.00	0.00	0.00	0.00
MCP	55.78	**58.67**	**65.30**	**72.44**	0.00	0.00	0.00	0.00
Eager T-replan	**53.23**	58.86	64.60	**69.31**	24.91	25.42	25.19	24.26
Reasonable T-replan	53.35	60.01	64.45	78.91	11.58	16.16	19.80	15.36
Lazy T-replan	53.57	**58.63**	**64.19**	71.11	7.08	28.05	37.57	25.58
Delay probability = 0.3								
	T=0	T=2	T=4	T=6	T=0	T=2	T=4	T=6
Eager T-All	651.97	122.80	101.04	100.55	0.00	0.00	0.00	0.00
Reasonable T-All	86.62	84.53	82.72	86.81	0.00	0.00	0.00	0.00
Reasonable T-Selective	65.04	69.28	75.15	81.79	0.00	0.00	0.00	0.00
MCP	**64.75**	**68.80**	**73.30**	**81.25**	0.00	0.00	0.00	0.00
Eager T-replan	**60.99**	67.73	**71.52**	**76.62**	29.23	37.23	40.85	41.47
Reasonable T-replan	61.14	68.08	72.17	80.16	15.23	23.77	25.59	37.44
Lazy T-replan	61.31	**66.91**	71.55	80.46	14.43	60.42	63.41	62.77

Table 9: Different replanning polices on instances with 10 agents.

7.4 Experimental Results

We performed a limited experimental evaluation of the different robust execution polices for our delayed communication on an 8x8 open 4-neighborhood grid with 10 agents. Delays were inserted randomly, with probability p per each move of each agent, where p is a parameter. We experimented with $p = 0.1, 0.2$, and 0.3. In this set of experiments the communication frequency was $T = 0, 2, 4$, and 6, and the original plans were built to match this frequency and thus were optimal and T-robust for $T = 0, 2, 4$, and 6, respectively. Table 9 presents our results, averaged over 50 instances. The columns report the execution costs (i.e., the overall sum-of-costs incurred until all agents reached their goals) and the CPU replaning runtime in ms required by the execution policies. The first four lines are for repair policies. The last three lines are for replan policies. Consider the repair policies. The results show that the $T - All$ policies have a higher cost than the $T - Selective$ policies. As expected, delaying all of the agents produce a higher cost than delaying only some of the agents so as to preserve their internal order. In addition, mostly the MCP policy had the lowest cost among all the repair policies. Among the replan policies, mostly the Eager T-replan policy had the lowest cost.

Overall we can see that executing with replan policies costs less than executing with repair policies and that the MCP policy results in a good trade-off between cost and replaning time. However, there is a tradeoff because replan incurs extra overhead as shown in the right side of the table. We have performed these experiments for other number of agents and the same trends were observed.

8. Related Work

There are several notable prior work that are related to kR-MAPF. We discuss these here briefly and describe the relation of each work to ours.

kR-MAPF can be viewed as a special case of *conformant planning* (Cimatti & Roveri, 2000; Brafman & Hoffmann, 2006), where the task is to find a plan that will be successful regardless of imperfect information about the initial state and action outcomes and without sensing capabilities.

kR-MAPF is different from Nguyen et al.'s (2017) robust planning, which is for planning and not MAPF, and address a setting in which an incomplete model of the domain is available and the task is to find a plan that is likely to succeed.

As part of their work on execution policies for MAPF with delays, Ma et al. (2017) also proposed a CBS-based algorithm that aims to minimize the expected makespan. Unlike our work, they assumed prior knowledge of delay probabilities and do not provide any guarantee on the returned plan. As describes earlier in this paper, they only produced MAPF-DP plans, which are identical to 1-robust plan.

Wagner et.al. (2017) proposed a MAPF variant that considers uncertainty called *MAPFU*. In MAPFU, there is uncertainty regarding the location (vertex) of each agent, including its initial and goal vertices. The probability for two agents to collide in some vertex can be measured using the given belief state of each agent. Given a threshold for each agent, a plan is called a solution if the probability of each agent to collide with the other agents is below its threshold. While in kR-MAPF the number of delays the agents experience is limited, in MAPFU the probability of each conflict to occur is limited, given the agent belief states.

One of the approaches we proposed for solving kR-MAPF is based on CBS (Sharon et al., 2015). Many improvements to CBS have been introduced throughout the years (Boyarski et al., 2015; Cohen, Uras, Kumar, Xu, Ayanian, & Koenig, 2016). Most can be applied on top of our k-robust CBS without further adjustments. These improvements can either enhance the kR-CBS algorithm, or modify the search in order to find suboptimal solutions.

An exception to these improvements is the *Meta-agent CBS* (MA-CBS) (Sharon et al., 2015) algorithm, where agents with many mutual conflicts are merged into a *meta-agent* that is then treated as a joint composite agent by the low-level solver. A k-robust version of MA-CBS requires a low-level solver that is also k-robust for meta-agents consisting two or more agents. This is a topic for future work.

In many cases in grid-based domains, each shortest path of one agent conflicts with each shortest path of other agent. This cause a rectangular area of conflicts. To decrease the size of the CT and enhance the search of CBS, Li et al. (2019a) suggested to identify this area and impose a large constraint on multiple cells (also called *Barrier Constraint*). As other CBS improvements, this also can (in the future) be adapted by range constraints to enhance the search of kR-CBS over grids.

A k-robust plan can be viewed as a plan in which each agent occupies more than one vertex each time step. Li et al. (2019b) suggested a MAPF solution for multiple large agents that occupy multiple vertices each time. These agents have a static shape and cannot occupy a sequence of vertices as the k-robust does. Atzmon et al. (2019) proposed a solution for multiple train-agents. While these agents occupy a fixed size sequence of vertices, in a k-robust plan whenever an agent performs a wait action it shrinks in a sense that it occupies

one less vertex. Therefore, the number of vertices that the agent occupies is dynamic and changes as a results of the number of wait actions it performed in the last k steps.

9. Conclusions and Future Work

In this paper we explored how to solve the multi-agent pathfinding problem in a way that is robust to unexpected delays. We proposed a robustness measure for MAPF plans called k-robust, where a k-robust plan is a plan in which each agent can experience up to k delays while still preserving the ability to follow the generated plan. Several ways to obtain k-robust plan were proposed, including solvers based on A*, CBS, and constraint programming (using Picat). Our experimental results show that finding a k-robust plan is possible and requires only a minimal increase in plan cost. In addition, our results also show that while the Picat-based solver was better in small domains, the CBS-based solver was better in large domains and solved successfully instances with up to 100 agents.

Then, we defined robust execution policies to be used online, when the agents are executing the plan. We proposed several classes of execution policies, compared their performance, and showed that using a k-robust plan for $k > 0$ as the baseline plan results in fewer re-plans during execution. Lastly, we explored how k-robust plans can be used in conjunction with the proposed execution policies to address cases where agents can only communicate every few time steps. We show in both cases the trade off between faster execution policies that repair the original plan and cheaper (lower cost) execution policies that replan from a given time and continue with a new plan.

There are many possible lines of future work, including: adapting other MAPF solvers such as the ICTS algorithm (Sharon, Stern, Goldenberg, & Felner, 2013) to find k-robust plans, finding k-robust plans for different objective functions (i.e. makespan), adjusting the execution policies for a non-uniform communication frequency, exploring ways to have different k values for each agent and between pairs of agents, and studying the impact of weighted actions.

Acknowledgements

Part of this research was done when Roni Stern was in Palo Alto Research Center (PARC), USA. This research was supported by the Israel Ministry of Science, the Czech Ministry of Education, by ISF grants #844/17 to Ariel Felner and #210/17 to Roni Stern, and by the Czech Science Foundation project P103-19-02183S to Roman Barták.

Appendix A. Proof of Observation 1

As a reminder, Observation 1 states that a plan is k-robust iff it does not contain any k-delay conflicts.

Proof. **Direction #1: Not k-robust → exists a k-delay conflict.** Assume that π is not k-robust. This means there is a set of delays $\mathcal{D}$ that includes at most k delays for each agent, such that $\mathcal{D}[\pi]$ is not valid. Since $\mathcal{D}[\pi]$ is not valid, there exists a pair of agents a_i and a_j and a timestep t such that $\langle a_i, a_j, t\rangle$ is a conflict in $\mathcal{D}[\pi]$. This conflict is either a

vertex conflict or a swapping conflict.

Case #1: $\langle a_i, a_j, t\rangle$ is a vertex conflict. This means that $\mathcal{D}[\pi_i](t) = \mathcal{D}[\pi_j](t)$. Since $\mathcal{D}$ contains at most k delays for each agent, there exists $\Delta_i, \Delta_j \in \{0, \ldots, k\}$ such that $\pi_i(t - \Delta_i) = \pi_j(t - \Delta_j)$. Without loss of generality, assume that $\Delta_i \geq \Delta_j$. Therefore, $\langle a_i, a_j, t - \Delta_i\rangle$ is a $(\Delta_i - \Delta_j)$-delay conflict in π. Since $\Delta_i - \Delta_j \leq k$, we have that π has a k-delay conflict, as required.
Case #2: $\langle a_i, a_j, t\rangle$ is a swapping conflict. This means that

$$\big(\mathcal{D}[\pi_i](t) = \mathcal{D}[\pi_j](t-1)\big) \wedge \big(\mathcal{D}[\pi_j](t) = \mathcal{D}[\pi_i](t-1)\big) \tag{2}$$

Since $\mathcal{D}$ contains at most k delays for each agent, there there exists $\Delta_i, \Delta_j \in \{0, \ldots, k\}$ such that

$$\big(\mathcal{D}[\pi_i](t) = \pi_i(t - \Delta_i)\big) \wedge \big(\mathcal{D}[\pi_i](t-1) = \pi_i(t - \Delta_i - 1)\big) \tag{3}$$

$$\big(\mathcal{D}[\pi_j](t) = \pi_j(t - \Delta_j)\big) \wedge \big(\mathcal{D}[\pi_j](t-1) = \pi_j(t - \Delta_j - 1)\big) \tag{4}$$

From Equations 2-4, we have that

$$\pi_i(t - \Delta_i - 1) = \pi_j(t - \Delta_j) \tag{5}$$

$$\pi_i(t - \Delta_i) = \pi_j(t - \Delta_j - 1) \tag{6}$$

If $\Delta_i = \Delta_j$ then $\langle a_i, a_j, t - \Delta_i - 1\rangle$ is a 1-delay conflict in π and thus π has a k-delay conflict, as required. If $\Delta_i \neq \Delta_j$, then assume without loss of generality that $\Delta_i > \Delta_j$. If $\Delta_j = 0$, then $\langle a_i, a_j, t - \Delta_i\rangle$ is a $\Delta_i - 1$-delay conflict due to Eq. 6. Since $\Delta_i \leq k$, this means π has a k-delay conflict, as required. Finally, if $\Delta_i > \Delta_j > 0$, then due to Eq. 5, we have that $\langle a_i, a_j, t - \Delta_i - 1\rangle$ is a $(\Delta_i + 1 - \Delta_j)$-delay conflict. Since $\Delta_j > 0$ and $\Delta_i \leq k$, then $(\Delta_i + 1 - \Delta_j) \leq k$ and thus π has a k-delay conflict, as required.
Direction #2: Exists a k-delay conflict $\rightarrow$ not k-robust. Let $\langle a_i, a_j, t\rangle$ be a k-delay conflict. By definition, there exists $\Delta \in \{0, \ldots, k\}$ such that $\pi_i(t) = \pi_j(t + \Delta)$. Let $D_i = \langle a_i, t + i\rangle$, and consider the set of delays $\mathcal{D} = D_0, \ldots, D_{k-1}$. Clearly, $\mathcal{D}[\pi]$ is not valid, having a vertex conflict at time $t + \Delta$, and thus π is not k-robust, as required. □

References

Atzmon, D., Diei, A., & Rave, D. (2019). Multi-train path finding. In *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 2019*, pp. 125–129.

Atzmon, D., Stern, R., Felner, A., Wagner, G., Barták, R., & Zhou, N. (2018). Robust multi-agent path finding. In *Proceedings of the Eleventh International Symposium on Combinatorial Search, SOCS 2018, Stockholm, Sweden, 2018*, pp. 2–9.

Boyarski, E., Felner, A., Stern, R., Sharon, G., Shimony, E., Bezalel, O., & Tolpin, D. (2015). Improved conflict-based search for optimal multi-agent path finding. In *IJCAI*.

Brafman, R. I., & Hoffmann, J. (2006). Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence*, *170*(6), 507–541.

Cimatti, A., & Roveri, M. (2000). Conformant planning via symbolic model checking. *J. Artif. Intell. Res.(JAIR)*, *13*, 305–338.

Cohen, L., Uras, T., Kumar, T. S., Xu, H., Ayanian, N., & Koenig, S. (2016). Improved solvers for bounded-suboptimal multi-agent path finding.. In *IJCAI*, pp. 3067–3074.

Erdem, E., Kisa, D. G., Oztok, U., & Schueller, P. (2013). A general formal framework for pathfinding problems with multiple agents. In *AAAI*.

Felner, A., Stern, R., Rosenschein, J. S., & Pomeransky, A. (2007). Searching for close alternative plans. *Autonomous Agents and Multi-Agent Systems*, *14*(3), 211–237.

Felner, A., Stern, R., Shimony, S. E., Boyarski, E., Goldenberg, M., Sharon, G., Sturtevant, N. R., Wagner, G., & Surynek, P. (2017). Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *the International Symposium on Combinatorial Search (SoCS)*, pp. 29–37.

Gange, G., Harabor, D., & Stuckey, P. J. (2019). Lazy CBS: implicit conflict-based search using lazy clause generation. In *International Conference on Automated Planning and Scheduling, ICAPS*, pp. 155–162.

Goldenberg, M., Felner, A., Stern, R., & Schaeffer, J. (2012). A* Variants for Optimal Multi-Agent Pathfinding. In *Workshop on Multi-agent Path finsing. Colocated with AAAI-2012*.

Hart, P., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*, 100–107.

Kornhauser, D., Miller, G., & Spirakis, P. (1984). Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In *Symposium on Foundations of Computer Science*, pp. 241–250. IEEE.

Lam, E., Bodic, P. L., Harabor, D. D., & Stuckey, P. J. (2019). Branch-and-cut-and-price for multi-agent pathfinding. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pp. 1289–1296.

Li, J., Harabor, D., Stuckey, P. J., Ma, H., & Koenig, S. (2019a). Symmetry-breaking constraints for grid-based multi-agent path finding. In *The Thirty-Third AAAI Conference on Artificial Intelligence, Honolulu, Hawaii, USA, 2019*, pp. 6087–6095.

Li, J., Surynek, P., Felner, A., Ma, H., Kumar, T. K. S., & Koenig, S. (2019b). Multi-agent path finding for large agents. In *The Thirty-Third AAAI Conference on Artificial Intelligence, Honolulu, Hawaii, USA, 2019*, pp. 7627–7634.

Ma, H., Kumar, S., & Koenig, S. (2017). Multi-agent path finding with delay probabilities. In *AAAI*.

Nguyen, T., Sreedharan, S., & Kambhampati, S. (2017). Robust planning with incomplete domain models. *Artif. Intell.*, *245*, 134–161.

Qin, W. B., Gomez, M. M., & Orosz, G. (2017). Stability and frequency response under stochastic communication delays with applications to connected cruise control design.. *IEEE Trans. Intelligent Transportation Systems*, *18*(2), 388–403.

Sharon, G., Stern, R., Felner, A., & Sturtevant, N. R. (2015). Conflict-based search for optimal multi-agent pathfinding. *Artif. Intell.*, *219*, 40–66.

Sharon, G., Stern, R., Goldenberg, M., & Felner, A. (2013). The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, *195*, 470–495.

Silver, D. (2005). Cooperative pathfinding. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pp. 117–122.

Spaan, M. T., Oliehoek, F. A., & Vlassis, N. (2008). Multiagent planning under uncertainty with stochastic communication delays. In *the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 338–345.

Standley, T., & Korf, R. (2011). Complete algorithms for cooperative pathfinding problems. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume One*, pp. 668–673. AAAI Press.

Standley, T. S. (2010). Finding optimal solutions to cooperative pathfinding problems. In *AAAI*.

Stern, R., Sturtevant, N. R., Felner, A., Koenig, S., Ma, H., Walker, T. T., Li, J., Atzmon, D., Cohen, L., Kumar, T. K. S., Barták, R., & Boyarski, E. (2019). Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 2019*, pp. 151–159.

Sturtevant, N. R. (2012). Benchmarks for grid-based pathfinding. *Computational Intelligence and AI in Games*, *4*(2), 144–148.

Surynek, P., Felner, A., Stern, R., & Boyarski, E. (2016). Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *ECAI*.

Surynek, P. (2010). An optimization variant of multi-robot path planning is intractable. In *AAAI*.

Surynek, P. (2012). Towards optimal cooperative path planning in hard setups through satisfiability solving. In *PRICAI*, pp. 564–576.

Wagner, G., & Choset, H. (2015). Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, *219*, 1–24.

Wagner, G., & Choset, H. (2017). Path planning for multiple agents under uncertainty. In *the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 577–585.

Wu, F., Zilberstein, S., & Chen, X. (2009). Multi-agent online planning with communication.. In *the International Conference on Automated Planning and Scheduling (ICAPS)*.

Yu, J., & LaValle, S. M. (2013a). Planning optimal paths for multiple robots on graphs. In *ICRA*, pp. 3612–3617.

Yu, J., & LaValle, S. M. (2013b). Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI*.

Zhou, N.-F., Barták, R., Stern, R., Boyarski, E., & Surynek, P. (2017). Modeling and solving the multi-agent pathfinding problem in picat. In *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*.

Zhou, N.-F., Kjellerstrand, H., & Fruhman, J. (2015). *Constraint solving and planning with Picat*. Springer.